RATIONALIZING APL

Kenneth E. Iverson
I.P. Sharp Associates
Toronto

Certain aspects of **conventional** APL (as defined in the IBM publication **APL Language** [1]) violate some of the fundamental characteristics of the language, and the definitions of some other aspects are too limited to provide clear guidance for systematic extensions. The present paper proposes convenient replacements for the anomalous facilities, and a systematic scheme for extensions, a scheme that does not invalidate the facilities defined in Reference 1, nor, indeed, those incorporated in many other systems, specifically SHARP APL.

The main aspects to be treated are:

A) Bracket and semicolon indexing, and indexed assignment.

B) Name assignment.

C) Function valence.

D) Function Definition.

E) Syntax and order of execution.

F) Extensions to higher rank arrays.

G) Operators on non-scalar functions.

H) Miscellaneous.

Items in the foregoing list can be used to illustrate the main point. For example, the indexing $A[I;J]$ cannot (without the use of ⍎) be applied to an array A of arbitrary rank, the list expression $(I;J)$ must be treated as a monolithic whole (i.e., not as a sequence of normal APL functions), and a name cannot be assigned to the resulting list. The reduction and inner and outer product operators are limited to scalar functions and could be extended in a variety of ways. For example, the reduction $F{\neq}A$ could (as it is in [2]) be assumed to apply F over each vector along the leading axis of A, or it could (as defined in [3]), apply F over the set of subarrays obtained by splitting A along the first axis. Both are consistent with the conventional definition, but only the latter allows the matrix product over a rank three array to be expressed as $+.\times{\neq}A$.

A section will be devoted to each of the eight items listed above, and additional sections will treat a number of new functions and operators defined according to the schemes developed in the earlier sections. Two appendices summarize the ranks assigned to existing primitive functions (to systematize all extensions to higher-rank arrays), and the definitions of dyadic operators.

Any problems attendant upon the use of implicit arguments such as ⎕*IO* are exacerbated by the more general uses of operators. Consequently, all new functions and operators to be defined will be **independent of ⎕*IO*, and will be defined in origin** 0. Moreover, 0-origin will be used throughout this paper.

Consider, for example, the operator ⍤ which we will define so that the function *G*←*F*⍤*I* applies *F* to its argument after transposing to the rear end those axes specified by *I*. If the operator is independent of ⎕*IO* (and uses fixed origin 0), then a function *G* that moves the leading axis to the rear is obtained from the expression *G*←*F*⍤0. However, if dependence on ⎕*IO* were to be adopted, which value of ⎕*IO* should apply, that in effect when *G* is specified, or when it is used? If the former, what advantage does the dependence offer, and if the latter, how could one specify the desired behaviour in the unknown environment?

The reader is assumed to be familiar with the functions **enclose**, **disclose**, and **match** (< > ≡) as defined in [4]. Moreover, we will assume that the symbol ⍬ denotes the constant <''; that is ⍬≡<''. This provides a convenient notation for a "fill" element appropriate to a non-simple array, just as 0 and ' ' provide fill elements for simple arrays. Moreover, it rationalizes the use of the symbol ⍬ in the outer product, so that the expression *A*.*B* may be used with *A*←⍬ (or *A*←<'') and *B*←×, just as it may be used with *A*←+ and *B*←×.

An interpreter for the language as extended here (written in APL) has proved invaluable in developing the design, and it has been used to produce the examples used herein. Any reader having access to this interpreter will find it helpful in studying the present paper.

An early version of the interpreter has been presented in [5]; except as it may prove useful in the treatment of language questions, the interpreter itself will not be discussed here.

To obviate repetitions of phrases such as "If *A* is an array of rank 3", we will adopt the following convention: *A*0, *B*0, etc., denote scalars, *A*1, *B*1, etc., denote vectors, and so forth.

A. INDEXING AND INDEXED ASSIGNMENT

The enclose function as defined in [4] has made it possible to produce by straightforward APL functions the "index lists" required in indexing expressions of the form $A[I;J]$, and therefore makes it possible to define a corresponding indexing function, which will be denoted by { and called **from**:

$I\{A \leftrightarrow A[\gt I[0];\gt I[1];\ ...]$

Since the disclose function > is permissive, the selection of any single element of A can be written without enclosures as, for example, 1 2 3{A3. Moreover, the left rank of { is 1 and its right rank is infinite, so that (as discussed in Section F) a simple left argument I of rank greater than 1 produces an array of shape $^{-}1\downarrow\rho I$ of elements chosen by the index vectors along its last axis, yielding what is sometimes called "scattered" indexing. For example:

(3 2 ⍴⍳6){A2 ↔ A2[0;1],A2[2;3],A2[4;5]

Each element $I[K]$ must be of the form S or <S or <<S, where S is a simple array. If $I[K]$ is of the form <<S, then the indexing along axis K is **complementary**, selecting all elements **except** those whose indices occur in >>$I[K]$. In particular, if >>$I[K]$ is empty, then the entire axis is selected.

In forming the left arguments of the indexing function, it will often be convenient to use the **link** function ⊃ defined as follows:

⊃B ↔ <B if B is simple
B if B is non-simple

$A\supset B \leftrightarrow (\lt A),\supset B$

For example, (2 3⊃4⊃⍬⊃5 6){A4 ↔ A4[2 3;4;;5 6].

The indexing function { as defined thus far provides all of the facilities provided by conventional indexing, and "scattered" and "complementary" indexing as well. Its power is further enhanced by allowing **negative** indexing (with the completely disclosed element E in position K, either >K{I or >>K{I, replaced by ($K\{\rho A)|E$) and **abbreviated** indexing (with $I\{A$ equivalent to (I,(0⌈(⍴⍴A)-⍴I)⍴<⍬){A). For example, 2 3{A3 ↔ (2⊃3⊃<⍬){A3 and, if ⍴A is 2 3 4, then 1 ¯1 1{A ↔ A[1;2;1]. Each index J along an axis of length N is limited by $(J\lt N)\wedge(N\geq -J)$.

Indexed assignment of the form $A[J;K]\leftarrow B$ is anomalous, or at least awkward; the result of the expression (as received by C in $C\leftarrow A[J;K]\leftarrow B$) is merely B, and the merged result normally wanted must be obtained by a separate reference to A.

What is desired is a merge of *A* and *B* controlled by an index *I* and by the particular selection function employed. A simple solution is provided by the **with** operator ⍤ (described in [6] as *A⍤F X* ↔ *A F X*) combined with the observation (made by Arthur Whitney) that a dyadic case of the derived function *I⍤*{ could be defined to produce the desired merge. Thus if:

R←*B I⍤*{*A*

then (⍴*R*)≡⍴*A*, and *B*≡*I*{*R*, and "the rest" of *R* agrees with the rest of *A*, Thus:

A2[*J*;*K*]←*B* ⋄ *A2*←⍉*A2* ↔ *A2*←⍉*B* (*J*⊃*K*)⍤{*A2*

In order to avoid the questions raised by expressions such as *X*[1 1]←2 3, it might be desirable to restrict the domain of the expression *B I⍤*{*A* to a "proper" index *I* containing no repeated elements. On the other hand, a permissive definition may be harmless, and could probably be implemented more easily.

B. NAME ASSIGNMENT

In conventional APL, the arrow used for assigning a name to a variable is not permitted for naming functions. Instead, a function name is assigned indirectly by characters embedded in its representation, and there is no direct way for assigning a name to a derived function such as +.×.

The solution is to allow the arrow to assign a name to any entity -- variable, function, or operator (as in *X*←⍳5 and *IP*←+.× and *RED*←/) -- and to provide means for function definition that do not embed the function name in its representation. This is discussed in Section D.

C. FUNCTION VALENCE

In conventional APL, functions are generally considered to be ambivalent, with the actual valence of application being determined by context. Many, however, are assigned a fixed valence -- some primitives (e.g., monadic ~ and dyadic ∈), most derived functions (with one exception, ⌽[*I*]), and all defined functions.

The introduction of ambivalent defined functions in many APL systems has proved very convenient to the user, and the admission of ambivalent derived functions opens up many new possibilities, one of which has been exploited in the use of -.×*M* for the determinant [7].

It is now proposed that (except for niladic functions, which must be treated separately), all functions be assumed to be ambivalent, and that any undefined case (such as ∈*X*) be treated as a proper function which has an empty domain. An unfortunate consequence of this is that **some change will occur** in the behaviour of existing programs, but only in replacing a syntax error message by a domain error message, or (as in the case of ~/3) by a result. However, this is a single, general change that would prevent further changes occasioned by each function extension introduced. Niladic functions must, of course, retain their fixed valence.

A further consequence of the ambivalence of all functions is that no operator can have its behaviour defined in terms of the valence of the function to which it is applied, since all have the same valence. This matter was anticipated in [4] in the definition of two distinct composition operators, one to apply a function *F* dyadically to the results of monadic applications of *G* (as in $\alpha\ F\ddot{\circ}G\ \omega \leftrightarrow (G\alpha)\ F\ (G\omega)$), and one to apply *F* monadically to the result of a dyadic application of *G* (as in $\alpha\ F\ddot{\circ}G\ \omega \leftrightarrow F\ \alpha G\omega$).

D. FUNCTION DEFINITION

Although the operators are the entities generally used to produce functions in APL, the only language facility (as opposed to the ∇-editing facility based upon it but not usable in programs) provided for producing defined functions (i.e., □*FX*), is itself a **function**, and one which produces as an explicit result only the **name** of the function, producing the function itself as a side-effect. Moreover, although many APL systems allow □*FX* to produce ambivalent functions, none provide true ambivalence, with independent localization of variables for the two cases.

The proposed replacement for □*FX* is a modification of the direct definition operator ∇ defined in [8], and is defined as follows:

1. *M*∇*D* produces a function, with *M* and *D* being the representations of the monadic and dyadic cases.

2. The general form of each representation is a vector *R* of enclosed **segments**, the segments being executed in an order determined by a **sequence control** vector. It is initially set to 1+⍳⍴*R* (the segment indices beginning at 1 as do the indices in conventional definition), and is reset by any branch statement having a non-empty argument. Termination occurs upon exhaustion of the sequence control vector, or upon an index out of range.

3. A label in the first element of *R* is assigned the value 1+⍳⍴*R*, the second is assigned 1↓1+⍳⍴*R*, etc.

4. The symbols ⍺ and ⍵ denote the left and right arguments, and Δ is used for **self-reference** to the function itself, being used in recursive definitions as well as for defining one of the two cases in terms of the other.

5. A name is localized if it occurs **immediately** to the left of an assignment arrow in any segment; for example, 3×*A*←4+*B* ←⍵ localizes *A* but not *B*. Name localizations for the monadic and dyadic cases are independent.

6. The explicit result of a function is the result of the last statement executed which produced an explicit result, where expressions such as *X*←3+4 or 3+4 are assumed to produce explicit results, but →*A* and ⍞'' and ⎕*A* (as defined in Section J) are not. Automatic output is **not** produced by an expression such as 3+4; such output is produced only by expressions using ⎕←.

7. Every vector *V* is treated as ,⊃*V*, that is, a simple vector is treated as a single segment. Single segment representations may therefore be written in the form '0⌈⍵' ∇ '⍺||⍵' .

A function produced by the operator ∇ may be assigned a name (as in *F*←*M*∇*D* or in *A*(*F*←*M*∇*D*)*B*), but it may also be used without assigning a name, as in *Y*←''∇'⍺+÷⍵'/*X*.

E. SYNTAX AND ORDER OF EVALUATION

Although the order of execution of functions and operators in conventional APL is well-defined, the order of **evaluation of names** is not, and ambiguity therefore occurs not only in an expression such as (*A*←2)÷*A*←3, but also (if *A* is a shared variable) in the simpler expression *A*÷*A*. The matter is further complicated by allowing the direct assignment of names, as in (*F*←3×⍳4)(*F*←×.-) *F*←⍳4.

Although the precedence of operators over functions is established in conventional APL, the fact that operators are not applied to derived functions implies that the syntax and order of execution of a sequence of operators is not established. The syntax and order of evaluation and execution that is now widely accepted (and is proposed here) may be summarized as follows:

a) Parentheses are obeyed as usual, and are permitted around function names, as in *A*(+)*B* and *A*(+.×)*B*.

b) Names are evaluated in order from right to left, and the values produced are detached from the original names.

c) An operator may be either monadic or dyadic, but is not ambivalent.

d) An operator applies to the result of the entire operator sequence to its left, much as a function applies to the result of the entire expression to its right. Thus

* +.×/*T* ↔ *(+.×)/*T*.

More precisely:

1. The tokens (i.e., the indivisible names such as *ABC* or ¯2.4*E*5 or + or ¯2.4*E*5 3 2) are examined in order from right to left; each is evaluated immediately and the representation of the result (whether variable, function, or operator) is appended at the beginning of a (**right**) **stack** vector and is immediately dissociated from the name whose evaluation yielded it.

2. The evaluation of tokens may be interrupted by the execution of a function or operator in the stack applied to one or two (or zero in the case of a niladic function) adjacent arguments in the stack.

3. The syntax and order of execution are further specified by a set of rules that determine what segment of the stack is to be executed, or whether evaluation of tokens is to proceed. The rules are based upon the class (variable, operator, function, assignment arrow) of each of the first four elements of the stack.

4. A right parenthesis invokes the execution of the entire expression up to the matching left parenthesis, the result being then transferred to the right stack.

5. Because conventional APL uses name assignment in the form *AB*←*X* rather than in the possible, but more awkward, form '*AB*'←*X*, rule 1 above must be modified as follows: if the leading element in the right stack is an assigment arrow, the next token is transferred directly **without evaluation**.

6. Items 4 and 5 conflict in the case (*N*)←3, a conflict resolved by giving precedence to item 4. The consequence (as elaborated in [5]) is to provide "indirect" assignment. For example, if *N*←'*ASD*' and (*N*)←3, then *N* retains the value '*ASD*', and *ASD* becomes 3.

7. The rules referred to in item 3 are summarized in the following table taken directly from the interpreter:

```
VMF←L   VF      D       VF      L S(1↑R),(R[2]RE R[1 3]),4↓R
VMF←L   V       F       V       L S(1↑R),(R[2]RE R[1 3]),4↓R
VMF←L   VF      M               L S(1↑R),(R[2]RE R[1]),3↓R
MF←L    F       V               L S(1↑R),(R[1]RE R[2]),3↓R
D       V       F       V       L S(2↑R),(R[2]RE R[3]),4↓R
V       ←       VMDF    R       L S,R[0] IS R[2]
L       VMDF    R               R[1]
VMDFR                           (>''ρZ)S(1↓Z←L LE 0),R
←                               (>''ρZ)S(1↓Z←L LE 1),R
                                R NOT SINGLE∈
```

Each of the first four columns above specifies the classes to which the corresponding element of *R* may belong to invoke the corresponding action specified by the final column. The symbols *V*, *M*, *D*, *F*, ←, *L*, and *R* denote the classes **variable, monadic operator, dyadic operator, function, assignment, left** (a filler introduced when the left stack is exhausted), and a **right filler** implicitly introduced to extend a short right stack to four elements.

Thus the first line specifies the action to be taken (replacing elements 1, 2, 3 of the stack by the result of applying element 2 [a dyadic operator] to elements 1 and 3) in the case that *R*[0] belongs to any one of the classes *VMF←L*, and *R*[1] belongs to *VF* and *R*[2] belongs to *D*, and *R*[3] belongs to *VF*.

No special treatment is required for a niladic function (represented in the usual way) since it is executed (exactly as a variable is) before being transferred to the right stack.

Further entries could be added at the head of the syntax table in order to catch invalid sequences in the right stack as soon as they occur, rather than allowing them all to appear as a single error due to the fact that the stack has the wrong form upon termination.

F. EXTENSIONS TO HIGHER RANK ARRAYS

In conventional APL, the scalar functions (which apply to scalar elements and produce scalar results) extend to higher rank arrays according to simple general rules; no corresponding general rules exist for the remaining so-called mixed functions.

For a general function *F*, the axes of an argument *ω* will be split at some point *K* such that *F* is applied to each subarray of shape *K*↓ρ*ω*, producing a result of shape (*K*↑ρ*ω*),*SIR*, where *SIR* is the (necessarily common) shape of the individual results produced by applying *F* to each subarray.

Each function *F* is assigned a monadic **rank** *MF* which determines the value of *K* as follows:

```
K←0⌈(ρρω)-MF        if MF≥0
K←(ρρω)⌊|MF         if MF<0
```

Consequently, a positive or zero value of *MF* determines the rank of the subarrays to which *F* applies, and a negative value determines the **complementary** rank, the number of axes excluded in each application of *F*.

Because of the definition of *K*, the rank of a function imposes only an upper limit on the rank of the subarrays to which *F* applies, and does not prohibit the application of *F* to **degenerate** cases of lower rank. For example, the function ⌹ has rank 2, but if its argument is a vector it is applied to the vector. Because the function ⌹ is defined on vectors, it yields a result, although another rank 2 function, such as the determinant, might be defined to yield a domain error in such a degenerate case.

Similar remarks apply to the dyadic case, with each function being assigned independent left and right ranks, and yielding outer shapes *KL*↑ρα and *KR*↑ρω. Unless one of these outer shape vectors is empty (in which case "scalar extension" applies, using the single subarray with each subarray of the other argument), these outer shapes must agree. The shape of the overall result is *S,SIR*, where *S* is the outer shape vector of greater length.

Certain functions, such as ravel, are of **infinite**, or **unbounded** rank; consistently with the foregoing definition, such a rank will be assigned an infinite value.

The utility of the notion of rank will now be illustrated briefly by introducing the **rank operator**, which yields a derived function of specified rank. Thus *F*⍤2 ¯1 1 yields a derived function of monadic rank 2, of complementary left rank 1, and right rank 1. Moreover, *F*⍤*R* ↔ *F*⍤3ρ*R*, so that abbreviated forms such as *F*⍤2 (all ranks are 2), and *F*⍤2 3 (all right ranks are 2) may be used. For example, ,⍤2ω ravels the last two axes of the argument, ,⍤¯2ω ravels all but the first two, and ,⍤0ω adds a final unit axis to the axes of the argument.

A necessary first step in establishing the use of ranks is to define the ranks of all existing functions. Because of the treatment of degenerate cases, it is possible to do this in a manner compatible with existing definitions by simply assigning infinite ranks to all. However, it is generally more useful to assign the lowest rank which will ensure compatibility with existing definitions, since this leads to simpler definitions of the functions, and greater utilization of the general scheme of

extension. In particular, the scalar functions can, and will, be assigned rank 0.

The argument for assigning to a function *F* the lowest compatible rank is somewhat weakened by the fact that a corresponding function of lower rank *R* is so easily obtained from the expression *F*⍤*R*. Thus, although ⌽ could be assigned monadic rank 1, it might instead be assigned infinite rank, so as not to make it dissimilar to ⍟, which must be assigned infinite monadic rank.

Appendix A shows a table of the proposed rank assignments, together with notes on the less obvious choices. The scalar functions are not included in the table.

G. OPERATORS ON NONSCALAR FUNCTIONS

The reduction operator can, as remarked in the introduction, be consistently extended to non-scalar functions in at least two different ways, and similar variety is possible for the remaining conventional operators. We will now present a systematic scheme of extension (based upon the notion of function rank) which applies to all of the existing operators except the (bracket) axis operator. This operator is excluded because of its anomalous syntax, and because its purpose can be conveniently served by the rank operator, sometimes in conjunction with the transpose operator.

The cases of reduction are straightforward: *F*⌿*ω* splits the argument *ω* along the leading axis and applies *F* between the resulting subarrays; *F*/*ω* splits along the final axis.

The outer product *α*∘.*Fω* splits the arguments at the points *KL* and *KR* as determined (in the manner described for *K* in Section F) by the ranks of *F*, and applies *F* between each pair of resulting subarrays, producing an overall result of shape (*KL*↑⍴*α*),(*KR*↑⍴*ω*),*SIR*.

The inner product *α*F.G*ω* splits *α* along the last axis and *ω* along the leading axis, applies the outer product ∘.*G* between each of the *N*←¯1↑⍴*α* (which must equal 1↑⍴*ω*) pairs of corresponding subarrays, and then applies *F* between the resulting outer products. It is defined more formally in Section K.

In order to remain completely compatible with conventional APL, the requirement that ¯1↑⍴*α* be equal to 1↑⍴*ω* could be relaxed to allow one or the other to be 1. However, it would be preferable to make the requirement strict.

A scan applies reduction over each of a set of partitions of one axis of the argument, and the result therefore includes a **supernumerary** axis corresponding to this set. The scan $F\backslash\omega$ partitions along the **last** axis, and places the supernumerary axis **last**, after the (necessarily common) shape of the individual reductions. The scan $F\backslash[\omega]$ partitions along the **first** axis, and places the supernumerary axis **first**, before the common shape of the individual reductions. Supernumerary axes are discussed more fully in Section K.

H. MISCELLANEOUS

This section treats three matters, a rationalization of the jot (∘) used in the outer product and of the **type** attribute of an array used in expansion and overtake, and some illustrations of the assertion (made in Section G) that the rank operator can fulfill the functions of the anomalous bracket operator.

The type attribute. For most purposes an APL array A is completely determined by its shape (which can be obtained by applying the function ⍴) and by the value of each of its elements (which can be obtained by indexing), and the array incorporates no other attributes which determine the result of applying functions to it. However, conventional APL includes two functions, the **overtake** (↑ with a sufficiently large left argument), and the derived function **expansion** ($U\backslash$, where U is Boolean) which do depend on an attribute associated with neither the value of an element nor the shape: if $N\leftarrow\iota 0$ and $C\leftarrow$'', then N and C agree in shape and differ in no element, but $0\backslash N$ and $0\backslash C$ produce different results (0 and ' '), as do $1\uparrow N$ and $1\uparrow C$.

This anomalous behaviour of overtake and expansion arose as follows: they are essentially "selection" functions, but they depend upon a parameter not explicitly specified by their arguments, namely, the **filler** quantity to be inserted in each of the "new positions" produced. Moreover the domain of the catenate function was limited to arguments of the same **class** (numeric or character) so as to preclude the introduction of heterogeneous arrays of mixed classes. Consequently, the filler could not be a fixed quantity, but had to be drawn from the same class as the argument of the overtake or expansion function.

For any argument A that contains at least one element (i.e., $1\leq\times/\rho A$), the filler can be determined from the class of any one of the elements, but for an empty array, no element can provide this information. The definition of overtake and expansion on empty arrays therefore required either the choice of a single filler for all, or the introduction of an attribute, usually referred to as the **type** of an empty array.

In conventional APL, arguments for the convenience provided by the type attribute (in maintaining the class of an argument in expressions such as *U\U/A* [for 0=∨/*U*] and *K↑J↓A* [for *J*≥⍴*A*]) won the day. Although compatibility requires that the type attribute be maintained to continue the present behaviour of expansion and overtake, it does not require any further extension of its use in the definition of new or existing functions and operators.

There are two classes of argument against further extension of the use of the type attribute of empty arrays: the complexity of the rules required for extension, and the possibility of providing the same (or nearly the same, or greater) convenience aimed at in the design of conventional APL, by means not then available.

Some of the complexity can be glimpsed from the following:

1. The need for rules for the types resulting from the expressions '',⍳0 and (⍳0),''.

2. The need for further type rules for any new classes that may be introduced, such as enclosed and heterogeneous [2,3].

Further indications of the complexity can be seen in the rather extensive discussions of types and prototypes in APL systems [2,3].

The introduction of dyadic cases for derived functions has made it easy to provide extra parameters, as in the merge produced by the expression *B I⍤⊂ A*. Moreover, an associated dyadic case of compression as discussed in Section K provides for the specification of fillers for expansion. Not only would 0 *U\ N* and ' ' *U\ C* specify the normal fillers for vectors *N* and *C* (empty or not), but 1 *U\ N* and '*' *U\ C* would provide useful generalizations.

Since **take** is a selection function, a dyadic case *B I⍤↑ A* could be defined to provide a merge analogous to *B I⍤⊂ A*. Moreover, the derived function *↑⍤A* (defined by *↑⍤A I* ↔ *I↑A*) could be provided by an associated dyadic case: thus *I ↑⍤A B* would employ a filler specified by *B*. Since the "new positions" need not even form a rectangular array, *B* should be a scalar, that is, the right rank of *↑⍤A* would be 0.

In conclusion:

1. Expressions such as *F U/⍵* and *I ⊂⍤⍵ F* should be introduced to permit convenient specification of a filler quantity *F*.

2. For any non-empty argument, a suitable filler should be defined for any new class of argument introduced. The filler <'' (denoted by ∘) will be used for enclosed arrays.

3. For empty arrays, compatibility requires the continuation of two distinct fillers, but no new fillers should be introduced. In particular, the result of 1↑0⍴<'A' should be defined as **zero**, with a view to eventually eliminating the type distinction and making **all** such fillers **zero**.

The bracket axis operator. The following remarks about the bracket axis operator are based upon unpublished suggestions by Arthur Whitney:

```
      +⍀⍤¯2⍵ ↔ +⍀[2]⍵ ↔ +⍀[2]⍵
      ⌽⍤¯2⍵ ↔ ⌽[2]⍵  ↔ ⊖[2]⍵
α,⍪⍤¯2⍵ ↔ α,[2]⍵
α,⍪<⍤¯2⍵ ↔ α,[1.5]⍵
```

In the foregoing identities, 0-origin indexing is assumed, and the identities do not hold for all degenerate cases of catenation that involve arguments of differing ranks. The function ,⍪ catenates along the leading axis, and ,⍪< produces lamination along a new leading axis. Note that lamination to introduce a new axis at any position can be achieved without the use of fractional indices.

I. NEW OPERATORS

This section treats only those new operators that will be used directly in the definition of new functions in Section J; others are deferred to Section K. To facilitate distinctions between the cases distinguished by argument types (function-function, function-variable, variable-function, and variable-variable), we will use the names *F* and *G* exclusively for functions.

This section treats three dyadic operators (denoted by ⍤, ⍥, and ⍪, and called **on**, **upon**, and **with**), although the cases *F*⍪*I*, and *I*⍪*F*, and *I*⍪*J* are deferred to Section K. It also treats the monadic operator ⍨, called **con**.

The monadic operator ⍨ is defined as follows: *α F*⍨*ω* ↔ *ω F α*, and the right and left ranks of *F*⍨ are therefore the left and right ranks of *F*. Moreover, the monadic case *F*⍨*ω* is the function inverse to *F*; the definition of this inverse (including its rank) is an **attribute** of the function *F*, that is, it is part of the definition of *F*. For example, the inverse of the scalar function * is the scalar function ⍟, and the inverse of the infinite rank function < is the rank 0 function >.

The name **con** (meaning **in opposition to**) is chosen to suggest both the function inversion of the monadic case, and the argument inversion of the dyadic case; the symbol ⍤ resembles the letter c, the initial letter of both con and commute.

The composition *F*⍤*G* is defined as in Section C, with the added proviso that the composition is **close** in the sense defined in Reference 4. In terms of function rank, this proviso may be stated very simply by declaring that the three ranks of the derived function *F*⍤*G* each equal the monadic rank of *G*.

The "dual" *F*⍨*G* has the same ranks as does composition, the definitions being (*G*⍤)*F* *G*⍵ and (*G*⍤)(*G*⍺)*F* *G*⍵. The derived function *F*⍥*G* has the same ranks as *G*.

The cases *F*⍤*I* and *F*⍥*I* and *I*⍥*F* will be defined so that together they provide the facility provided by the axis operator defined in Reference 4, but in a simpler and more flexible manner.

The case *F*⍤*I* was defined in Section F, where it was referred to as the **rank** operator. The case *F*⍥*I*⍵ applies *F* to a transpose of ⍵ that moves axes *I* to the tail end. For example, if ⍴⍵ is 3 4 5 6 7, then ⍴⊢⍥2 1⍵ is 3 6 7 5 4, and ⍴⊢⍥0⍵ is 4 5 6 7 3. (The identity function ⊢ is defined in Section J.)

In the general case, the right argument of ⍥ has three elements, one to specify the transpose to be applied to each of the argument cases: monadic, left, and right. Each is enclosed, and a simple or abbreviated argument is extended as follows: *F*⍥*I* ↔ *F*⍥(3⍴⊃*I*). Negative indexing of **axes** can be used, as, for example, in ⊢⍥¯2⍵ to interchange the last two axes.

The case *I*⍥*F* is defined similarly, except that the indicated axes are moved to the **front**. In conjunction with the identity function, the operator ⍥ can, as illustrated above, provide transpositions that would be much more awkward to write using the transpose function ⍉.

Derived functions produced by the function definition operator ∇ are defined to have infinite rank.

J. NEW FUNCTIONS

This section defines a number of new functions, most of which have been proposed in earlier papers. The main point is to illustrate the ease of definition provided by the joint use of direct definition and rank specification. The symbol ¯ is used to denote infinity:

Name	Symbol	Definition
Dex	⊢	'⍵'∇'⍵'
Lev	⊣	''∇'⍺'
Nub/Cup	∪	'((⍵⍳⍵)=⍳⍴⍵)/⍵←,⍵'∇'⍺,⍵' ⍤¯1 1
Member	∈	'(∪⍵)∘.(⍙⍤0)⍵'∇'∨/⍺≡⍤0,⍵' ⍤¯1 0
Onub/Cap	∩	'(,⍤0⍙∪⍵){∪⍵'∇'(⍺∈⍵)/⍺' ⍤¯1 1
Not	~	'1-⍵'∇'(⍙⍺∈⍵)/⍵' ⍤0 1 1
CP/From	{	*M*←'>∘.(,⍤¯1 0 1)⍤¨>/(<⍤>⍵),∘' *D*←'>(⍳0)⍴⍤1(*S*⊥⍤1(*S*←*N*↑⍴⍵)|⍤1⍙⍺)⊤⍤¯1 0 1,<⍤(-*N*←⍴,⍺)⍵' *M*∇*D*⍤1 1 ¯

The monadic case defined for ∈ is the **distribution** function defined in [6]; the function ≡ used in its definition is defined in [5], and yields a 1 if its arguments agree. The dyadic case of ~ is the set difference function.

The monadic case of { is the **Cartesian** product over the enclosed elements of its argument, and yields a result whose shape is the catenation of the shapes of (⍴⍤¨>⍵),<⍴⍵. For example:

```
      { 1 2 ⊃ 3 4 ⊃ 5
1 3 5
1 4 5

2 3 5
2 4 5
```

The self-reference (⍙) in the definition of the dyadic case is used to define the **from** function discussed in Section A; except for the complementary indexing provided by the use of doubly enclosed elements, this definition is complete.

K. FURTHER OPERATORS

The operators discussed in this section are not essential to the rationalization program for existing functions and operators carried out in Sections A-H. They do, however, constitute important extensions, and introduce two further notions: the placement of "supernumerary" axes produced in scans and similar "partitionings" along an axis, and the "coherence" of a function, which can range from infinite for a function *F*, to zero for the outer product ∘.*F*. They also help to illuminate the mnemonic schemes employed in the choice of operator symbols and in the disposition of operator arguments.

Because of the large number of functions required in a language, the adoption of mnemonic schemes (such as the use of a single symbol for related monadic and dyadic cases, for application to arrays of any rank and class, real or complex, as

well as the use of a single operator with existing functions to obtain sums, products and maxima over arrays) can be an important aid to the user. Because a large number of operators will also be required, care should also be exercised in assigning symbols to them.

Since each argument of a dyadic operator may belong to either of two classes (function or variable), and since each derived function produced is ambivalent, each operator symbol can produce as many as eight classes of functions; various kinds of similarities can be employed to group these functions in mnemonically useful ways.

For example, the symbol ⍤ is called **on**, and the first entry in Appendix B shows the use of different interpretations of the meaning of "on": the case *F*⍤*G* is the application of *F* on the result of *G* (called **composition** in mathematics); the case *F*⍤*I* is the application of *F* on each of the pieces obtained by splitting the argument into subarrays of ranks determined by *I*; the case *I*⍤*F* is the application of *F* on pieces obtained by splitting the argument **along** an axis or axes in a manner determined by *I*; and *I*⍤*J* is the **constant function** whose value is *I*, applied on subarrays of ranks determined by *J*.

In cases such as *F*⍤*I* and *I*⍤*F*, it is preferable to assign *F*⍤*I* to the more commonly used case, because *F* itself will often be a derived function, and an expression such as *I*⍤(+.×) requires parentheses, whereas +.×⍤*I* does not.

The symbol ⍥ in the second entry of Appendix B is called **upon**: the case *F*⍥*G* denotes application of the monadic function *F* upon the result of the dyadic function *G*, the larger circle (as compared to ⍤), indicating the larger valence in the application of *G*; the cases *F*⍥*I* and *I*⍥*F* concern the application of *F* upon the result of a transpose (a function that also employs ⍉ in its symbol), the transpose moving the axes designated by *I* to the front if *I* precedes the operator, and to the back if it follows.

Some of the relations among the cases of the operator denoted by the dot can be appreciated only by understanding that the inner product *F*.*G* is (except for a transposition of the left argument) a very special case of **tensor contraction** produced by combined use of the monadic and dyadic cases of the form *F*.*I*, as in +. 2 ⍺ ×. 2 ⍵, which "runs together" the first two axes of ⍺ and ⍵ (forming an outer product between the rest), and then applies + reduction over axis 1 and axis 0, in that order. The equivalence of *F*. 0 and ∘.*F* has already been remarked. For example:

```
R←(2 3 4⍴⍳24)×. 2(2 3 5⍴⍳30)
```

```
    ρR
2 3 4 5

    -. 2 R
¯300 ¯312 ¯324 ¯336 ¯348
¯315 ¯327 ¯339 ¯351 ¯363
¯330 ¯342 ¯354 ¯366 ¯378
¯345 ¯357 ¯369 ¯381 ¯393

    -⌿-⌿⍤¯1 R
¯300 ¯312 ¯324 ¯336 ¯348
¯315 ¯327 ¯339 ¯351 ¯363
¯330 ¯342 ¯354 ¯366 ¯378
¯345 ¯357 ¯369 ¯381 ¯393
```

Til. Appendix B shows two cases for the **til** operator, denoted by }. The case *F*}*I* ω is the *I*th power of *F*, and *F*}-*J* ↔ *F*⍣}*J*, so that negative values of *J* give powers of the inverse function. Moreover, if *I*≥0, then *F*}*I* has rank *MF*; if *I*<0 it has the rank of the inverse function *F*⍣¯1.

The second case *F*}*G* has ranks (*MG*⌈*RF*),*RF*, and *MG*, and is defined as follows:

$$\alpha\ F\}G\ \omega \leftrightarrow (G\omega)F\alpha$$
$$F\}G\ \omega \leftrightarrow (G\omega)F\omega$$

As remarked in [9], the case α *F*}*G*}*H* ω is equivalent to (*G*α)*F*(*H*ω) and is therefore very useful. It was also stated that α*F*}⊢ω is equivalent to α*F*⍣ω; although true in some instances, this is not true in general because the rank of the identity function ⊢ (which is infinite) may differ from the left rank of *F*.

Dot. The dot operator, conventionally defined only for the case *F*.*G*, is extended to the case *F*.*K*. The monadic case *F*.*K* ω is defined as *K* successive reductions of the form *F*⌿⍤*R*. For example: *F*. 3 ω ↔ *F*⌿ *F*⌿⍤¯1 *F*⌿⍤¯2 ω. It should be noted that this definition ensures that the reduction must take place over the *K* successive axes, whereas any one of a succession of reductions of the form *F*/ *F*/⍤1 *F*/⍤2 might (because of the characteristics of the arbitrary function *F*) entirely change all axes of its argument.

The dyadic case α*F*.*K*ω provides a generalization of the outer product α∘.*F*ω in which the non-negative integer *K* determines how many of the leading axes of α and ω must agree, and in which the remaining outer axes form an outer product. For example:

```
    R←(A←2 3 4⍴⍳24)⌻. 2 (B←2 3 5⍴⍳30)
    ρR
2 3 4 5
```

```
      0 0{R
 0  0  0  0  0
 0  1  2  3  4
 0  2  4  6  8
 0  3  6  9 12

      R≡ 0 1 2 0 1 3⍴⍳A∘.×B
1
```

Formally, the dyadic derived function *F.K* is said to have **coherence** *K*, and in the expression α *F.K* ω, the first *B*←0⌈*K*⌊((⍴⍴α)-*LF*)⌊(⍴⍴ω)-*RF* axes of α and ω are said to be **bound**, and must agree. Moreover:

$$\rho\alpha\ F.K\ \omega \leftrightarrow (B\uparrow\rho\alpha),(B\downarrow(-LF)\downarrow\rho\alpha),(B\downarrow(-RF)\downarrow\rho\omega),SIR$$

where *SIR* is the shape of each individual result produced by *F*.

If *G*←*F.K*, then *K* is (like the argument ranks) a **primary** attribute of the function *G* in the sense that it must be known to apply *G* (dyadically) to **any** argument. It is therefore desirable to assign an appropriate value of this **coherence** or **binding** attribute to every function, including the primitives.

Assignment of infinite coherence to all primitives would be compatible with the present definitions, including cases of scalar extension. It would, however, generalize the notion of scalar extension (in a useful manner), since it would make valid expressions such as (?2 3 4⍴9)+2 3⍴9.

As noted in the introduction, the monadic and dyadic cases of *F.K* together provide the contraction used in tensor analysis.

The conventional case of the inner product (*F.G*) was defined only for scalar functions *F* and *G*. A general definition for arbitrary functions can now be stated as:

$$\alpha\ F.G\ \omega \leftrightarrow F\neq(\bar{}1\ddot{\circ}\vdash\alpha)G.\ 1\ \omega$$

The left and right ranks of *F.G* are infinite.

Supernumerary axes. Certain operators produce derived functions that may introduce one or more **supernumerary** axes in addition to the axes produced by the particular function to which the particular operator is applied. These supernumerary axes may be defined either to precede or follow the normal result axes of the function, the whole collection following, of course, the outer axes of the argument to which the derived function is applied.

For example, scan introduces one extra axis contributed by applying reduction to each of a set of partitions of the

argument, and a general derivative operator applied to a function of rank R introduces R extra axes (for differentiation with respect to each element of the argument).

Scan. Each scan operator behaves like the corresponding reduction operator, and places the supernumerary axis first if the split (in the reduction) is along the first axis, and last if the split is along the last axis. This definition provides conformity with conventional APL.

Partition Operator. With an integer left argument K, the operator ⍤ provides a family of derived functions such that $K⍤F$ applies F to pieces of its right argument, the pieces being determined by the left argument or, in the monadic case, by the right argument itself. The right ranks of all cases are infinite.

There are seven cases: K=0 provides the general case, in terms of which each of the others can be expressed rather easily. Its left rank is 2, and F is applied to a rectangle of size |1{⍺ beginning at 0{⍺ or, more precisely, to the rectangle R selected as follows:

$$R←((((¯1↑⍴⍺)↑⍴⍵)|0{⍺)+⍤1⍤1>⍳⍤1⍤1>|1{⍺){⍵$$

Before applying F, the piece R is reversed along each axis for which the specified size (that is, 1{⍺) is negative.

Note that both negative and abbreviated indexing may be used in 0{⍺ to specify the position of the selected rectangle, and that reversal along any set of axes may be specified by negative elements in 1{⍺.

The degenerate case of a vector left argument ⍺ is treated as 0 1∘.×⍺, so as to select a piece of size ⍺ beginning at the origin. The monadic case $0⍤F$ ⍵ is equivalent to ((⍴⍴⍵)⍴⌊/⍴⍵)$0⍤F$ ⍵, that is, it selects the "maximal cube" from ⍵.

The cases ⍺ $1⍤F$ ⍵ and ⍺ $¯1⍤F$ ⍵ both apply F to pieces obtained by splitting ⍵, along the leading axis, into segments **beginning** at each 1 in the boolean vector left argument ⍺, the elements at the point of partition being excluded in the case of $1⍤F$, and included in the case $¯1⍤F$. The supernumerary axis introduced is placed before the individual results. A "short" left argument ⍺ is extended cyclically by (⍴⍴⍵)⍴⍺.

The left ranks of the functions $1⍤F$ and $¯1⍤F$ are 1, and extension to higher ranks is made in the usual way. The monadic cases are equivalent to the dyadics with a left argument of (0{⍵)≡⍤¯ ¯ ¯1 ⍵, that is, splits begin at occurrences of the "delimiter" specified by the leading subarray of ⍵.

The cases $2\ddot{\circ}F$ and $\bar{}2\ddot{\circ}F$ differ from the foregoing only in that each 1 in the left argument determines the **end** of a segment, and that the delimiter in the monadic case is the **last** subarray of ω, namely, $\bar{}1\{\omega$.

The remaining cases, $3\ddot{\circ}F$ and $\bar{}3\ddot{\circ}F$, provide (possibly overlapping) tesselations. The left rank is 2, and the expression $\alpha\ 3\ddot{\circ}F\ \omega$ applies F to each **complete** rectangle of size $|1\{\alpha$ obtained by beginning at all positions obtained as integer multiples of (each element of) the **movement** vector $0\{\alpha$. As in the case $0\ddot{\circ}F$, reversal of each piece occurs along the axis for which the "window" $1\{\alpha$ is negative.

The degenerate case of a vector α is equivalent to the left argument $1\ 1\circ.\times\ \alpha$, and therefore provides a complete, non-overlapping, tesselation of size α. The monadic case provides tesselation by "maximal cubes" having the size $(\rho\rho\omega)\rho\lfloor/\rho\omega$.

The case $\bar{}3\ddot{\circ}F$ differs from $3\ddot{\circ}F$ only in that any "shards" of sizes less than $|1\{\alpha$ are included.

With. If the ranks of the arguments I and J in the expressions $I\ddot{}F\ \omega$ and $F\ddot{}J\ \omega$ are equal to the left and right ranks of F, then the definitions are straightforward:

$$I\ddot{}F\ \omega \leftrightarrow I\ F\ \omega$$
$$F\ddot{}J\ \omega \leftrightarrow \omega\ F\ J$$

However, arguments I and J of greater rank introduce supernumerary axes which are placed before the individual results. Consequently, the derived functions behave more like outer products of F than like direct applications of F. For example, if $A\leftarrow 1\ 2\ 3$, then $A\ddot{}+A \leftrightarrow A\circ.+A$.

Compression. The definitions of the derived functions $I/$ and $I\not$ (called **compression** or **replication**) are based upon a vector argument I. The effect of $I\not\omega$ is to split ω along its **first** axis, select subarrays determined by I, and array them along a supernumerary axis that is placed **first**; the effect of I/ω is to split along the **last** axis and to place the supernumerary axis **last**.

An argument I of rank greater than 1 will introduce further supernumerary axes; they are placed **first** in both I/ω and $I\not\omega$.

Dyadic cases of $I/$ and $I\not$ should also be defined, with negative values in I providing selection from the left argument [10].

L. SUMMARY

The proposals in this paper essential to the rationalization program described in the introduction can be summarized as follows:

1. The enclose function < provides lists which are used as arguments to a straightforward indexing function ({, called **from**) to provide "scattered", "complementary", "abbreviated", and "negative" indexing, as well as all of the facilities provided by conventional indexing. The dyadic case of $I\ddot{}\{$ provides a merge that obviates indexed assignment.

2. The extension of ← to allow any APL entity to its right provides convenient name assignment for functions and operators as well as variables.

3. The definition operator ∇ provides convenient definition of independent monadic and dyadic cases of functions.

4. The syntax rules are extended to operators (to give them "long left scope") by a table shown in Section E. The recognition of cases, and the corresponding evaluations, involve only the first four elements of the execution stack.

5. A uniform scheme for extending functions to higher-rank arrays is provided by assigning to each function *F* four primary attributes that determine its application to any arguments: a monadic rank *MF*, a left rank *LF*, a right rank *RF*, and a coherence *CF*.

   In the normal case of arguments of sufficiently high rank, the function applies to the "low-order" subarrays of appropriate rank to produce individual results of a common shape *SIR*, the shape of the overall result of $\alpha\ F\ \omega$ being:

   $$(CF\uparrow\rho\alpha),(CF\downarrow(-LF)\downarrow\rho\alpha),(CF\downarrow(-RF)\downarrow\rho\omega),SIR$$

   The conformability imposed on the arguments is that $CF\uparrow\rho\alpha$ must equal $CF\uparrow\rho\omega$.

APPENDIX A

RANKS OF PRIMITIVE FUNCTIONS

The following table shows the proposed ranks for all primitive APL functions except the scalar functions, whose ranks are all zero. The ranks are shown in the order monadic, left, and right, with an overbar denoting infinite rank. The indices in the left column refer to notes following the table. The notes employ *A*0 and *B*0 to denote scalars, *A*1 and *B*1 to denote vectors, etc.

The proposed ranks were chosen in the light of the following general considerations:

1. **Infinite**, or **unbounded** rank will always allow compatibility with existing definitions, since the **degenerate** cases (of arguments having rank less than the maximum imposed by the stated rank of the function) can be defined as desired. Wherever necessary, infinite rank is chosen.

2. The minimum possible rank is preferred because it allows the simplest definition of the function, the definition for arguments of higher rank following from the general extension rule.

3. Questions concerning the possibly differing shapes of individual results (which, for example, precluded the inclusion of dyadic query (*?*) among the scalar functions) are ignored because **close composition** can be used to produce individual results of a common shape. For example, *A*<⍤?*B* produces individual scalar results which are enclosed vectors of (possibly) differing lengths.

4. Because of anomalies in its syntax and its definition, the "bracket axis operator" is not included among operators whose definitions depend upon the ranks of their function arguments. Consequently, the definitions of expressions involving the bracket operator are left strictly unchanged, and their behaviour does not figure in the choice of ranks made here.

5. Because of anomalies in its syntax and its definition, the "bracket indexing function" is not included among the functions to which operators apply, and no attempt will be made to define its rank.

NOTE	*F*	*M*	*L*	*R*
	<	¯	0	0
	>	0	0	0
1	⍴	¯	1	¯
2	,	¯	¯	¯
3	⌽	1	1	2
	⍟	¯	¯	¯
4	⍉	¯	1	¯
5	↑↓		1	¯
6	⍳	1	1	0

NOTE	*F*	*M*	*L*	*R*
	∈	¯	0	¯
	⍋⍒	¯	¯	¯
7	?	0	0	0
8	⍷	2	2	2
9	⊥		1	¯
	⊤		¯	¯
	⍕		¯	¯
10	⍎	1		
	≡	¯	¯	¯

Ranks of Primitive Functions
Table 1

Notes

1. The degenerate case of *A*0⍴*B* is defined as in conventional APL.

2. Ranks 1 1 for dyadic could have been considered because they work for all cases of the form *A*2,*B*2 and *A*3,*B*3, but they would not agree with the present definition of cases such as *A*2,*B*1 and *A*3,*B*2.

3. Ranks 0 1 for dyadic would be preferable except that this does not agree with the present permissive use of *A*1⌽*BK*, since it would yield an additional 1 at the beginning of the shape vector of the result. Ideally this definition should be changed; if this is not possible, the ranks 1 1 2 may be used, since they allow the special treatment of the (then) degenerate case. Either choice clearly shows the unit difference in rank required of arguments to the function.

4. The degenerate case *A*0⍉*BK* will remain as it is.

5. The degenerate cases *A*0↓*B*1 and *A*0↑*B*1 will remain as in conventional APL.

6. Rank 0 for the monadic case would introduce an unmanageable anomaly in the present permissive treatment of the case ⍳*A*1←,*A*0 since the result would necessarily have an outer shape of 1.

   Moreover, rank 1 would allow (or even suggest) the extension to the "odometer" function such that ⍳*A*1 yields a result of shape *A*1,⍴*A*1 containing (along its last axis) all indices of an array of shape *A*1. The present anomalous treatment of the case where ⍴*A*1 is 1 could be maintained.

   For the dyadic case, the ranks ¯ 0 would allow the production of vector indices to an array of any rank. For example, (3 3⍴⍳9)⍳2 3 4 would (in 0-origin) produce the matrix 3 2⍴0 2 1 0 1 1. However, this would imply that the indices to a vector should produce an array of one-element vectors, and that the shape of *A*1⍳*AK* would be (⍴*AK*),1 rather than ⍴*AK* as it now is. Compatibility could, of course, be obtained by treating a vector left argument as a special degenerate case.

7. Rank 0 0 is preferred for the dyadic case, but may be precluded by the permissive use of one-element vector arguments, since it would make the cases *A*0?*B*1, *A*1?*B*0, and *A*1?*B*1 differ from *A*0?*B*0 by an added 1 in the shape vector. However, it is possible that the use of the deal function is as yet infrequent enough to permit the necessary change in its definition.

   The choice of rank 1 1 would make it possible to continue the present anomalies for degenerate cases. Although this choice makes the function less versatile, it should be noted that the "correct" behaviour would be easily obtained in the derived function ?⍤0.

8. Since *A*2⌹*B*2 ↔ (⌹*B*2)+.×*A*2, the result is a collection of results for the **columns** of *A*2, and a left rank of 1 cannot, therefore, be compatible with the conventional definition.

9. Because ⊥ applies to vectors along the leading (rather than the trailing) axis of the right argument, a bounded right rank cannot give compatible results.

10. Treating execute as a normal function of rank 1 could be very useful. For example, if *A*2 is numeric, then ⍎*B*2←⍕*A*2 would yield *A*2. However, since the arguments of ⍎ may be expressions which produce side-effects, and since the order of applying any function to its cells is not prescribed, the results could be unpredictable.

APPENDIX B

TABLE OF DYADIC OPERATORS

The following conventions are used:

I and *J* denote variables, and *F* and *G* functions.

MF, *LF*, and *RF* denote monadic, left, and right ranks of *F* and *AF* denotes all ranks of *F*.

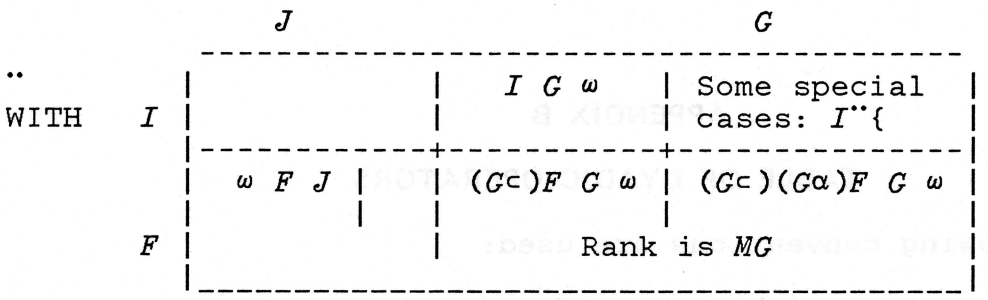
The monadic derived function is shown to the left of the dyadic.

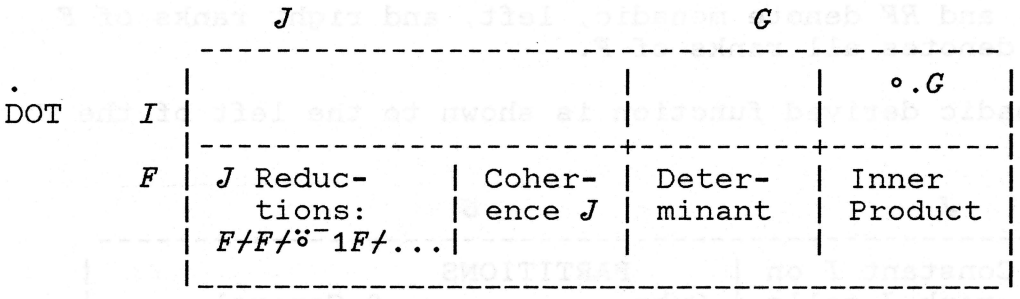
⍤ ON

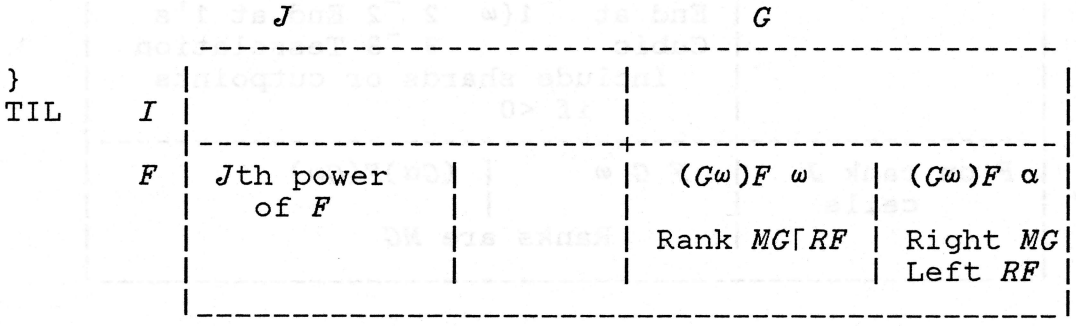
	J	*G*	
I	Constant *I* on rank *J* cells	PARTITIONS Cube 0 General Begin at 0{⍵ 1 ¯1 Begin at 1's End at ¯1{⍵ 2 ¯2 End at 1's Cubic 3 ¯3 Tesselation Include shards or cutpoints if <0	
F	*F* on rank *J* cells	*F G ⍵* | (*G*⍺)*F*(*G*⍵) Ranks are *MG*	

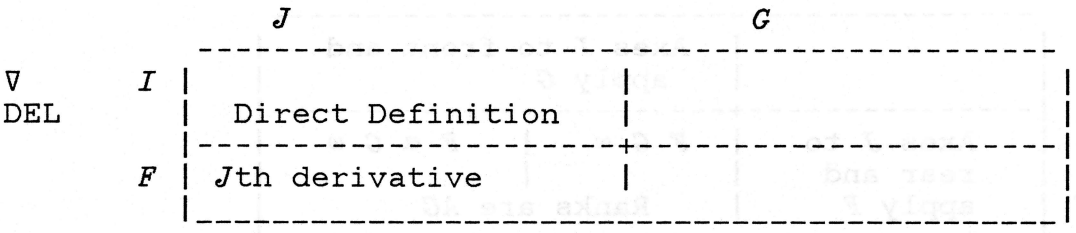
⍥ UPON

	J	*G*	
I		Axes *I* to front and apply *G*	
F	Axes *J* to rear and apply *F*	*F G ⍵* | *F* ⍺ *G* ⍵ Ranks are *AG*	


J
G
..
WITH
I
F
I G ω
Some special cases: I¨{
ω F J
(G⍤)F G ω
(G⍤)(Gα)F G ω
Rank is MG


J
G
.
DOT
I
F
∘.G
J Reduc-tions: F⍁F⍁⍤¯1F⍁...
Coher-ence J
Deter-minant
Inner Product


J
G
}
TIL
I
F
Jth power of F
(Gω)F ω
(Gω)F α
Rank MG⌈RF
Right MG
Left RF


J
G
∇
DEL
I
F
Direct Definition
Jth derivative

REFERENCES

1. **APL Language** (IBM GC26-3847).

2. NARS

3. APL2.

4. Robert Bernecky and Kenneth E. Iverson, 1980.

5. APL83

6. Operators and Functions - RC7091, 1978.

7. Discussion of the Determinant,

8. Direct Definition, Proceedings of APL81, San Francisco (ACM, New York).

9. Arthur Whitney and Kenneth E. Iverson, "Practical Uses of a Model of APL", **Apl Quote Quad**, Volume 13, Number 1, (Proceedings of APL82, Heidelberg), ACM, New York.

10. Eugene E. McDonnell, "Mesh and Mask", I.P. Sharp Associates, Palo Alto.